COMSC-210 Lab 11
Priority Queues, Trees, and Heaps

GOOD PROGRAMMING PRACTICES *show / hide*

ABOUT THIS ASSIGNMENT
In this assignment you will write a template class to implement a priority queue, and run timing tests to confirm O(log n) for its push and pop operations. It is based on the code developed in our lectures.

After you complete this lab assignment, post the required files to the COMSC server so that you receive the proper credit for this 50-point lab. Your posted files are due at midnight of the evening of the due date indicated in the course outline. Use the "Submit your FA2015 work" link on the class website to post your file for this lab, using the **lab11** folder provided.

LAB 11a: Write And Test A Priority Queue Template `[ PriorityQueue.h and PriorityQueueDriver.cpp ]`
Purpose. In this lab you will write your own priority queue template, using a dynamic array (not DynamicArray).

Write **`PriorityQueue.h`** and **`PriorityQueueDriver.cpp`**, based on the lecture notes -- NOT the textbook. Write `PriorityQueueDriver.cpp`, to fully test your templated class' functionality.

Name the template `PriorityQueue`. Use the algorithms developed in lecture (reproduced at the end of the lecture topic 11 notes). Include only these in the public interface: `PriorityQueue(int=2)`, `void enqueue(const T&)`, `T& front()`, `T& back()`, `T dequeue()`, `bool empty() const`, `void clear()`, and `int size() const`, with these exact prototypes. The defaulted parameter in the constructor is the initial capacity.

Store values in a dynamic array (not your DynamicArray template). Include the three memory management functions, because the template uses dynamic memory.

Either include a private function or a code block to double capacity if needed when `enqueue` is called. Do NOT include code to halve or otherwise reduce the capacity, because that will affect big oh confirmation of the `dequeue` function in the next part of this assignment.

Submit the two files (1 CPP and 1 H) to the class website for credit.

Program I/O. Input: All hard-coded in the driver CPP only. Output: From the driver CPP only, console (`cout`) output only.

LAB 11b: Perform A Simple Timing Study `[ PriorityQueueBigOh.cpp ]`
Purpose. In this lab you will apply the "4-cycle timing code" presented in the lecture topic 8 notes. The purpose is confirm that the big oh for enqueue and dequeue in your priority queue implementation are both O(log n).

Requirements. Write **`PriorityQueueBigOh.cpp`**, applying the "4-cycle timing code" from the lecture

notes, to the following functions, based on a priority queue of `doubles`, `ints`, `chars`, or `strings` -- your choice:

1. enqueue, O(log n)
2. dequeue, O(log n)

This will require *two* sets of "4-cycle timing code", each with their own starting `n` values. In each cycle, create the priority queue and fill it with numbers n down to 1 -- this is the fastest way to create the queue, because no swapping should take place. Then start the timer, perform the enqueue or dequeue operation (using "reps"), and stop the timer. In the timed enqueue loop with reps, add the values n+1 up to n+reps, so that maximum swapping takes place, forcing the log(n) behavior. Refer to lecture topic 8's notes for timing of fast operations, because these O(log n)'s are fast.

HINT: Remember the limitation on REPs with the operation being tested changes the n of what you're testing!

HINT: Start with a PriorityQueue object with an initial capacity high enough to avoid ever calling the doubling function. Put `assert(false);` in your capacity doubling (and halving, if you have any) code so you are *sure* you are not doing any doubling. Just be sure to remove it when you are done testing.

Submit the CPP file to the class website for credit.

Program I/O. No input. Console output reporting clearly labeled timing results per the provided code.

Example (using a computer that can measure time in milliseconds.).

```
Priority Queue enqueue, O(log n)
20685 (expected O(log n)) for n=8000000
21129 (expected 21587) for n=16000000
22283 (expected 22489.1) for n=32000000
23400 (expected 23391.1) for n=64000000

Priority Queue dequeue, O(log n)
21954 (expected O(log n)) for n=8000000
28176 (expected 22911.4) for n=16000000
32904 (expected 23868.7) for n=32000000
35033 (expected 24826.1) for n=64000000
```

Dequeue actually gives worse than O(log n) results. Think about the idealistic assumption made about the binary tree when computing dequeue's big oh. Think about the "spherical chicken in a vaccum in outer space" analogy. Using REPS for our timing process, see if you can explain why dequeue's big oh performs worse than expected.

NOTE: Be sure to use assertions to verify that (1) the queues contain the right numbers of entries (that is, n before the timing clock starts, and n+REPS after the clock stops), and (2) the values in the queue are ordered properly. To do the latter, you can dequeue all values from the queue after the clock stops, and test that each successive value is less than the one before it.

NOTE: If your computer only measures time in 1/60 second increments, this will not work very well for you -- see the next note.

NOTE: because conditions are not ideal (not always a full tree, n changes, multi-tasking CPU, etc.) you may not see good results. So the best you may be able to do is *infer* the result. If you can show that (1) it's

not O(n) and (2) the PQ values are ordered properly, then it must be working. And that's enough to confirm O(log n) in this case.

How to pause a console program: *show / hide*

GRADING RUBRIC *show/hide*
